Name: **Batch:** **Roll No:**

CS2005 Test I, Part I: Data Structures and Algorithms, Jan. 2017

1. Can a function $f(n)$ be $\theta(n)$ and $\theta(n^2)$ at the same time? Justify. [2]

Solution: $f(n) = \theta(n) \Rightarrow$ we can find c_1, c_2, n' such that $c_1 n \le f(n) \le c_2 n, \forall n \ge n'$. $f(n) = \theta(n^2) \Rightarrow$ we can find c_3, c_4, n'' such that $c_3 n^2 \le f(n) \le c_4 n^2, \forall n \ge n''$. From the two inequalities, we get

$$c_3 n^2 \le f(n) \le c_2 n, \forall n \ge n_0 (= \max(n', n'')) \Rightarrow n \le c_2/c_3, \forall n \ge n_0$$

which is never true. Hence we have a contradiction and hence $f(n)$ cannot be $\theta(n)$ and $\theta(n^2)$ at the same time

2. A double ended Queue (dequeue) supports insertion and deletion at the front and back. The dequeue support six fundamental functionalities.

- $InsertFirst(Q, e)$ inserts element e at the beginning of the dequeue Q.
- $InsertLast(Q, e)$ inserts element e at the end of the dequeue Q.
- $RemoveFirst(Q)$ removes the first element of the dequeue Q.
- $RemoveLast(Q)$ removes the last element of the dequeue Q.
- $First(Q)$ and $Last(Q)$ returns the first and last element (without any deletions) of the queue, respectively.

A doubly linked list implementation (with a node defined as **struct** $node\{element, left, right\}$, where $left$ and $right$ are pointers to type $node$) can implement a dequeue with all operations taking $O(1)$ time.

(a) Show the implementation of $InsertFirst(Q, e)$, $RemoveLast(Q)$ and $Last(Q)$ using a doubly linked list taking $O(1)$ time, using the notation that $Q.head$ points to the front node of the dequeue and $Q.tail$ points to the last node of the dequeue. [Pay special attention towards cases when the dequeue is empty (null) and when dequeue has just one element.] [5]

Solution:

function INSERTFIRST(Q, e) ▷ e is not a node, but an element
    $nodeptr \leftarrow$ create new $node$
    $nodeptr \rightarrow element \leftarrow e$
    $nodeptr \rightarrow left \leftarrow NULL$
    $nodeptr \rightarrow right \leftarrow NULL$
    if $Q.head = NULL$ **then**
        $Q.head \leftarrow nodeptr$
        $Q.tail \leftarrow nodeptr$ ▷ This is important
    else
        $nodeptr \rightarrow right \leftarrow Q.head$
        $Q.head \rightarrow left \leftarrow nodeptr$ ▷ Important that you point the ▷ previous first to the current first.
        $Q.head \leftarrow nodeptr$

Solution:

function REMOVELAST(Q) ▷ You don't specify the input
    if $Q.tail = NULL$ **then**
        return Underflow Error

> **if** $Q.tail = Q.head$ **then** ▷ Important to reinitialize both of them to NULL
> $tempnode \leftarrow Q.tail$ ▷ $tempnode$ is a pointer to the node to which ▷ $Q.tail$ was pointing to.
> $Q.tail \leftarrow NULL$
> $Q.head \leftarrow NULL$ ▷ Important
>   **free** $tempnode$ ▷ Important to free the memory corresponding ▷ to the node which was deleted
> **else**
> $tempnode \leftarrow Q.tail$
> $Q.tail \leftarrow Q.tail \rightarrow left$
>   **free** $tempnode$ ▷ Important
> $Q.tail \rightarrow right \leftarrow NULL$ ▷ Important that the new tail is pointing ▷ to NULL and not pointing to the deleted node.

> **Solution:**
> **function** LAST(Q)
>   **if** $Q.tail = NULL$ **then**
>   **return** Underflow Error
>   **else**
>   **return** $Q.tail \rightarrow element$

(b) Adapter Patterns are implementations of a given Abstract Data Type (ADT) using the functionalities of another ADT. The functionalities of a stack are $isEmpty()$, $Top()$, $Push(e)$, $Pop()$ (where $Top()$ returns the top of the stack without deletion and other functionalities are as defined in the lectures). Implement this stack using a *dequeue* defined above, with each operation taking $O(1)$ time. [Assume that you already have an $isEmpty()$ function corresponding to the dequeue, which you can use during the implementation of the stack.] 4

> **Solution:**
> **function** ISEMPTY( )
>   **return** ISEMPTY(Q)
> **function** TOP( )
>   **return** $Last(Q)$ ▷ If the Q is empty, then underflow error ▷ would be returned by $Last(Q)$
> **function** PUSH(e)
> $InsertLast(Q, e)$
> **function** POP( ) ▷ Remember that Pop has to delete and *return* the top element
>   **if** ISEMPTY( ) **then**
>   **return** Underflow error
>   temp = TOP( )
>   REMOVELAST(Q)
>   **return** temp

3. *Different Implementations, different efficiencies, different needs.*

We have seen the implementation of a priority queue using heaps. We have seen the running time of a priority Q operations using a heap are Insert: $O(\lg n)$, Find-Max (without deletion): $O(1)$ and Extract-Max (with deletion): $O(\lg n)$. Now consider the implementation of a priority Q using a (reverse) sorted (linked) list.

(a) Argue the runtime complexities of the best implementation of Insert, Find-Max and Extract-Max using the sorted list, without putting out any actual pseudo-code. $\boxed{3}$

Solution:

- *Insert* takes $O(n)$ as we have to traverse the sorted list, until we find an element which is less than the inserted element.
- *Find-Max* would take $O(1)$ time, as the maximum value will always be in the head of the list, which is independent of the length of the list.
- *Extract-Max* would take $O(1)$ time, as we are always deleting from the head of the list, which is independent of the length of the list.

(b) Assuming your application has more calls to Find-Max and less number of Insert calls, which of the two implementations of the Priority Q would you prefer and why? $\boxed{1}$

Solution: Since both the approaches have the same runtime complexity for Find-max, the decision would be based on the complexity of inserts. Since Insertion complexity is higher for the sorted list, we would go for the Heap. (However, if we had more Extract-Max than Find-Max, then we would choose the sorted list.)